

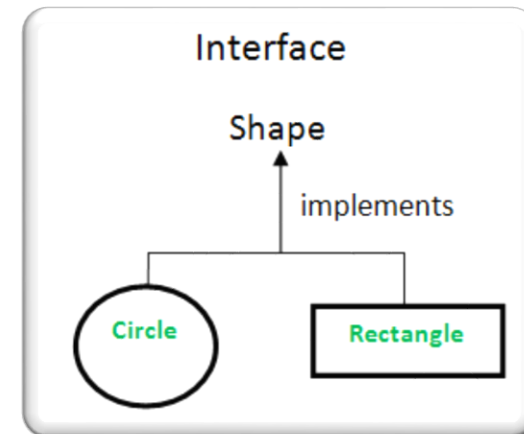
[Estructuras de Datos]



```
package youtube;
/**
 * @author Ericka Zavala
 * @version 1.0
 */
public class Ericka
{
    private final boolean suscrito, mesajes;
    public Ericka(boolean suscrito, boolean mesajes)
    {
        this.suscrito = suscrito;
        this.mesajes = mesajes;
    }
    public void Mensaje()
    {
        if(suscrito && mesajes)
            System.out.println("Mensaje");
        else
            System.out.println("No hay mensaje!");
        System.out.println("¡¡¡¡¡ Suscrito!!!");
    }
}
```

MANEJO DE EXCEPCIONES EN JAVA

Genericidad



CONCEPTOS BÁSICOS SOBRE EL MANEJO DE EXCEPCIONES.
GENERICIDAD PARAMÉTRICA
INTERFACES.

Copyright

- Copyright © 2019 Ing. Federico Joaquín (federico.joaquin@cs.uns.edu.ar)
- Las presentes transparencias constituyen una guía acotada y simplificada de la temática abordada, y debe utilizarse únicamente como material adicional o de apoyo a la bibliografía indicada en el programa de la materia.

CONCEPTOS BÁSICOS SOBRE MANEJO DE EXCEPCIONES

¿Qué son las excepciones?

- Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa.
 - Puede ser el resultado de una condición de error o una entrada incorrecta.
- En Programación Orientada a Objetos (POO), las excepciones son objetos lanzados por un código que detecta una condición inesperada, y es capturada por otro código que debe reaccionar ante esta situación intentando repararla.

Tipos de excepciones

CHECKED

Son excepciones que **obligan** tanto al **servidor** de un servicio como al **cliente** de un servicio, a **tomar medidas** frente a estas.

La clase **servidor**, en caso de lanzar excepciones, debe **indicar** dicha situación en la **signatura** del método que la lanza.

La clase **cliente**, al utilizar un **servicio** que indica que se puede **lanzar excepciones**, debe capturarlas y reaccionar ante estas, o debe propagarla indicando esto también.

UNCHECKED: RUNTIME EXCEPTIONS

Son excepciones que **no obligan** a las clases **cliente** de un servicio a **tomar medidas** frente a las mismas.



En esta materia solamente **se permitirá** el uso de excepciones del tipo **checked**.

Tipos de excepciones: unchecked

- Considere el siguiente fragmento de código:

```
public class Calculadora {  
    public static float dividir(int dividendo, int divisor){  
        return dividendo/divisor;  
    }  
  
    public static void main(String [] args){  
        float resultado = dividir(10, 0);  
        System.out.println("El resultado es: " + resultado);  
    }  
}
```

- El compilador no indica problema alguno.
- En ejecución, al intentar **dividir 10 por 0**, se provoca una **situación excepcional**, que Java modela mediante una ArithmeticException.
- Note que esta excepción, al no ser capturada, implica que el programa aborte.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Calculadora.Calculadora.dividir(Calculadora.java:6)  
at Calculadora.Calculadora.main(Calculadora.java:10)
```

Tipos de excepciones: checked

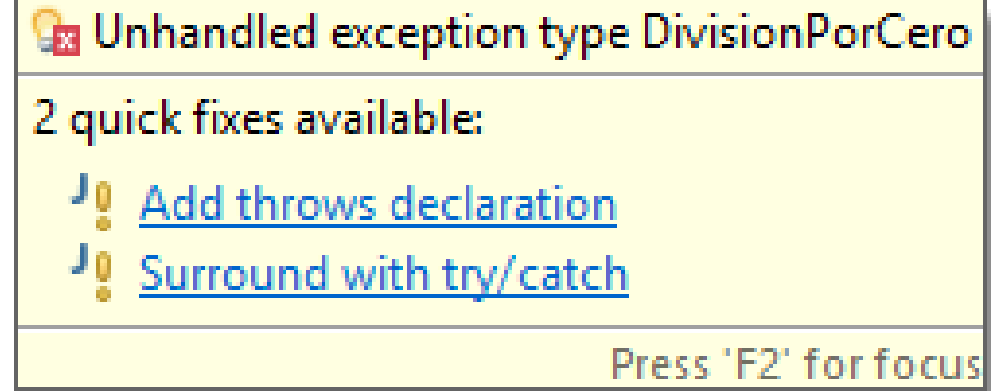
- Considere ahora que al fragmento de código anterior, se le agrega funcionalidad para que el método dividir chequee que el divisor sea distinto de cero, y en caso contrario, lanse una excepción notificándole al cliente de dicho error:

```
public class Calculadora {  
    public static float dividir(int dividendo, int divisor) throws DivisionPorCero{  
        if (divisor == 0) throw new DivisionPorCero("Error");  
        return dividendo/divisor;  
    }  
}
```

- Note que ahora la clase servidor, especifica en su signatura que puede lanzarse una excepción del tipo DivisionPorCero.
- Cualquier clase cliente que utilice el método dividir(...), está obligada a tomar medidas frente a esta situación.
 - Puede capturar la excepción, o bien, propagarla.

Tipos de excepciones: checked

```
public class Calculadora {  
    public static float dividir(int dividendo, int divisor) throws DivisionPorCero{  
        if (divisor == 0) throw new DivisionPorCero("Error");  
        return dividendo/divisor;  
    }  
  
    public static void main(String [] args){  
        float resultado = dividir(10, 0);  
        System.out.println("El resultado es: " + resultado);  
    }  
}
```



- Tal cual está el código, el compilador indica un error en la llamada al método dividir(...).
- Eclipse nos permite posicionarnos por sobre la línea de error, y nos ofrece un conjunto de soluciones rápidas disponibles.
 - Capturar la excepción con Try-Catch
 - Propagar la excepción.

Tipos de excepciones: checked

- Capturar la excepción con Try-Catch

```
public static void main(String [] args){  
    float resultado = 0;  
    try {  
        resultado = dividir(10, 0);  
    } catch (DivisionPorCero e) {  
        System.out.println(e.getMessage());  
    }  
    System.out.println("El resultado es: " + resultado);  
}
```



¿Qué sucederá cuando se captura la excepción? ¿Y cuando la excepción es propagada?

- Propagar la excepción.

```
public static void main(String [] args) throws DivisionPorCero{  
    float resultado = dividir(10, 0);  
    System.out.println("El resultado es: " + resultado);  
}
```

Tipos de excepciones: checked

- Capturar la excepción con Try-Catch

```
Error  
El resultado es: 0.0
```

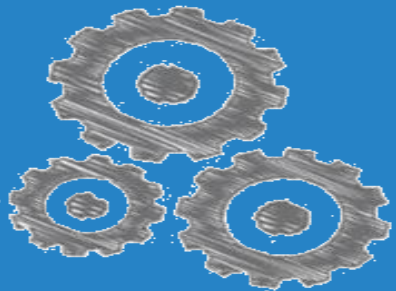
- Propagar la excepción.

```
Exception in thread "main" Calculadora.DivisionPorCero  
at Calculadora.Calculadora.dividir(Calculadora.java:5)  
at Calculadora.Calculadora.main(Calculadora.java:10)
```



¿Qué sucederá cuando se captura la excepción? ¿Y cuando la excepción es propagada?

Problema para analizar...



Manejo de excepciones

- Imagine un sistema para modelar transacciones bancarias, en donde dos clientes (ClitA y ClitB), van a transferirse dinero entre sí. La clase Banco dispone de un método transferir(...) que recibe dos clientes, y una cantidad de dinero a transferir desde la cuenta del primer cliente hacia la cuenta del segundo cliente.
- La clase Cliente cuenta con los métodos extraer(...) en donde se indica cuánto dinero extraerle al cliente, y un método depositar(...) que indica la cantidad de dinero a incrementar al mismo.
- Estos métodos lanzan las excepciones SinFondosException y SuperaMaximoException respectivamente, en el caso que al extraer no existan fondos en la cuenta, o cuando al momento de depositar, el dinero que se acumula en la cuenta supera el máximo permitido para el cliente en cuestión.

Problema para
analizar...



Manejo de excepciones

- Dada la siguiente implementación de transferir(...) en la clase **Banco**, ¿qué errores en cuanto a la funcionalidad detecta?

```
public class Banco {  
    public void transferir(Cliente CltA, Cliente CltB, float monto){  
        try {  
            CltA.extraer(monto);  
        } catch (SinFondosException e) {  
            System.out.println(e.getMessage());  
        }  
        try {  
            CltB.depositar(monto);  
        } catch (SuperaMaximoException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```



No siempre una excepción puede manejarse imprimiendo un mensaje. Por el contrario, ante estas situaciones se deben tomar las medidas necesarias para garantizar la correctitud y robutez de un programa.

GENERICIDAD PARAMÉTRICA EN JAVA

Repaso: Genericidad Pre Java 5

- Una alternativa para admitir colecciones genéricas en las versiones de Java anteriores a la 5, era mediante el uso de colecciones de tipo Object.

- Por ejemplo, declarando un arreglo de elementos Object:

```
private Object [] coleccion = new Object[10];
```

- Naturalmente, esto presentaba serias dificultades en el manejo de los datos:

- No se puede garantizar homogeneidad de los elementos.
- Se deben realizar castings explícitos a la hora de recuperar elementos.

```
coleccion[0] = new Integer(10);  
coleccion[1] = new String("Hola que tal");
```

```
Integer a = (Integer) coleccion[0];  
String b = (String) coleccion[1];
```

Genericidad Paramétrica: Java 5 y posterior

- En las versiones de Java 5 y posteriores, se incluye genericidad paramétrica.
 - Un tipo genérico es un tipo de dato que es definido en tiempo de ejecución, pero que permite definir una clase en función de parámetros formales de tipo que abstraen el tipo de dato interno que en ejecución será instanciado.

```
public class Encapsulador <E> {  
    private E elemento;  
  
    public E getElemento(){ return elemento;}  
    public void setElemento(E e){ elemento = e;}  
}
```

- Note que la clase Encapsulador, está totalmente definida en términos de un tipo de dato E que se desconoce en tiempo de compilación (puede ser cualquier tipo de dato).
- La clase almacena un elemento E, retorna un elemento E, modifica un elemento E, lo que permite verificar la correctitud de tipos en todo momento.

Genericidad Paramétrica: Java 5 y posterior

- En las versiones de Java 5 y posteriores, se incluye genericidad paramétrica.
 - Un tipo genérico es un tipo de dato que es definido en tiempo de ejecución, pero que permite definir una clase en función de parámetros formales de tipo que abstraen el tipo de dato interno que en ejecución será instanciado.

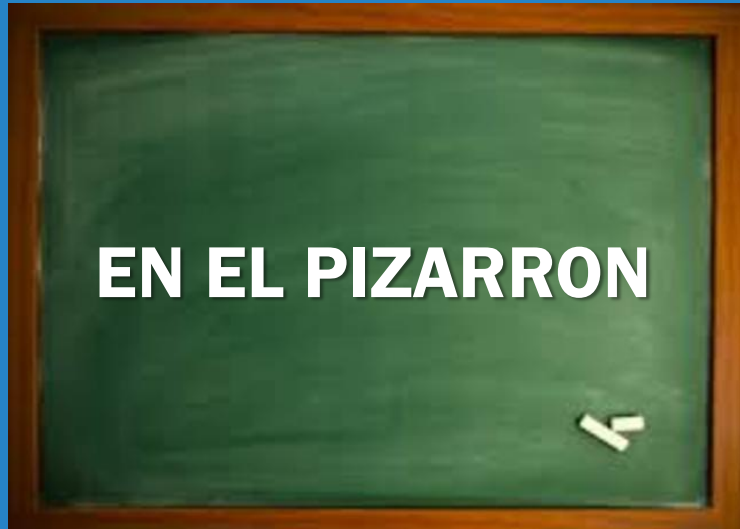
```
public class Encapsulador <E> {  
    private E elemento;  
  
    public E getElemento(){ return elemento;}  
    public void setElemento(E e){ elemento = e;}  
}
```

- Note que en tiempo de ejecución, el tipo genérico E fue instanciado con Integer y Float para las clases referenciadas por encapInt, encapFloat respectivamente.

```
public static void main(String [] arg){  
    Encapsulador<Integer> encapInt;  
    Encapsulador<Float> encapFloat;  
  
    encapInt = new Encapsulador<Integer>();  
    encapInt.setElemento(10);  
  
    encapFloat = new Encapsulador<Float>();  
    encapFloat.setElemento(10.5f);  
  
    System.out.println("Entero: " + encapInt.getElemento());  
    System.out.println("Float: " + encapFloat.getElemento());  
}
```


INTERFACES EN JAVA

Problema propuesto.



Colección de elementos

- Defina una interfaz colección de elementos genéricos que permita las siguientes operaciones:
 - void insertar(elemento e)
 - boolean pertenece(elemento e)
 - void eliminar(elemento e)
 - void mostrar()
- Desarrolle una clase ColeccionConArreglo, que implemente una colección de elementos genérica, almacenando internamente los elementos en un arreglo.
- Desarrolle una clase ColeccionConVector, que implemente una colección de elementos genérica, almacenando internamente los elementos en un vector.

Para analizar...



Colección de elementos

- ¿Qué ventaja existe desde el punto de vista del cliente el haber definido una interfaz?
- ¿Qué hubiese sucedido si en lugar de definir una interfaz colección de elementos, se hubiesen implementado las clases ColeccionConArreglo y ColeccionConVector directamente?

```
public static void main(String [] args){
    Coleccion<Integer> colInt = ColeccionConArreglo<Integer>();

    colInt.insertar(10);
    colInt.insertar(15);

    if (colInt.pertenece(15)) colInt.eliminar(15);
    else
        System.out.println("El numero 15 no pertenece a la colección");
}
```



El haber definido una interfaz Colección, permite que colInt pueda ser usado como una colección sin importar cómo realmente está implementada. Por ejemplo, es posible instanciar colInt de la siguiente forma, sin modificar el código restante: `colInt = ColeccionConVector<Integer>();`

Para tener en cuenta...



Tipos de datos abstractos (TDA)

- En Estructuras de Datos, nos basaremos en el concepto de interfaces para definir tipos de datos, y estudiaremos las diferentes alternativas existentes para implementar dichos tipos de datos.
- Es importante tener en cuenta que el TDA es quien define el conjunto de datos y operaciones que dicho tipo admite.
 - En el ejemplo anterior, indicamos que una Colección almacena elementos genéricos e, independientemente de cómo esté implementada, permite insertar, eliminar, consultar pertenencia y mostrar los elementos que almacena.

Para tener en cuenta...



Tipos de datos abstractos (TDA)

- De una forma más informal, un **TDA** define un **conjunto de datos** y **qué** se puede **hacer** con dichos datos.
- Las **implementaciones** de estos **TDA**, nos dicen **cómo** es que los **datos** se **organizan**, y **cómo** las **operaciones** se **llevan a cabo**.
 - En el ejemplo anterior, vimos **dos implementaciones** de una **colección** utilizando internamente un **arreglo** o un **vector** respectivamente.



Fin de la presentación.